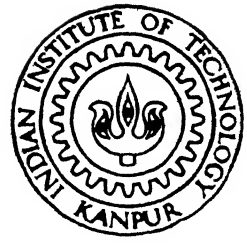
A JDBC Meta-driver for Remote Database Applications

by

Naga Sridhar Kataru



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

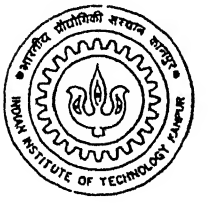
APRIL, 1998

A JDBC Meta-driver for Remote Database Applications

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by

Naga Sridhar Kataru



to the

Department of Computer Science & Engineering
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

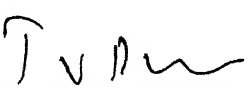
April, 1998

CSE-1998-M-KAT-J DBC

Entered in system

Certificate

Certified that the work contained in the thesis entitled "*A JDBC Meta-driver for Remote Database Applications*", by Mr.*Naga Sridhar Kataru*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.



(Dr. T. V. Prabhakar)
Professor,
Dept. of Computer Science and Engineering,
IIT Kanpur.

April, 1998

Abstract

In recent times, there is a growing need for organizations to access accurate information quickly and regardless of the data source, platform or database location. JDBC API has been designed to address this problem. But, in providing database access through JDBC, the issue of accessing remote databases has been completely left to the JDBC driver-developer. This project is an attempt to relieve the driver-developer of the task of explicitly implementing remote access mechanisms in the driver. A software component, called "Meta-driver", which incorporates these mechanisms has been designed and implemented for this purpose. Meta-driver is logically situated above the driver-layer and below the application layer. The developer simply includes this component to provide location transparency of databases.

In addition, by means of Meta-driver, one can easily develop Java applications for use with remote DBMSs. It provides seamless access to all remote database systems involved, and offers convenience to the users. It can be easily attached to the newly being-designed drivers or existing drivers(JDBC/ODBC), to make them useful for remote database applications.

Acknowledgments

The greatest debt of gratitude, I reserve for Dr. T. V. Prabhakar who guided me at every stage of this project with his perspicacious suggestions, who has been paternalistic towards me throughout my stay here, whose qualities like forbearance and patience influenced me a lot.

I am also beholden to Dr. Sumit Ganguly, for helping me out in many situations. And I extend my sincere thanks to Dr. D. Manjunath for being the examiner of my Defense.

I wish to express my heart-felt gratitude to Mr. Brahmaji Rao, for helping me solve the perennial problems encountered during changing of PCs and in administration. I would like to specially thank Ravi Kiran for abetting me in securing required literature for this project.

I am greatly indebted to my friends Rajesh, Dhanabal, Yugandhar, Vihari, Vivek Ranjan and others, for being always affectionate to me, despite my juvenile delinquency. Finally, I wish I could express my thankfulness in words, to my parents, sister and grand father, for their love and affection I have been receiving.

Contents

List of Figures

Chapter 1

Introduction

It is a common phenomenon that academic and industrial organizations continue to amass huge amounts of data as the time passes. Generally, that data is stored and managed by a Database Management System(DBMS).

When it comes to DBMS world, there is a plethora of DBMS vendors to choose, offering DBMSs with various capabilities and utilities. It has been observed that, as more data appeared, more DBMSs were purchased. Since migrating all the data into one system was Herculean task, many people preferred running two or more database systems independently. Programmers were forced to write different applications for different DBMSs, since each system had different protocols, Application Programming Interface(API), and data structures.

This might seem pointless, because, at a certain level, most database APIs offered the same features. They usually implemented most of the features of the standard SQL. Moreover, to be competitive, organizations need to access accurate information very quickly, regardless of the data source, platform or location.

So, management put the pressure on in-house developers to invent database clients for each employee's desktop machine that can find all the company's information from one application. This was the impetus for designing an API to provide database independence for developers.

The answer to this call was Open Database Connectivity (ODBC). Developed by Microsoft and based on the Call Level Interface specification of the SQL Access

Group, ODBC allows users to access data in heterogeneous environments of relational and non-relational databases. Developers, using the ODBC API, can write one application to access data on many vendors' DBMSs on many platforms, including Windows, Macintosh, and UNIX. A variety of vendors supply ODBC driver managers and drivers and drivers for these platforms.

But, with the advent of Java, the scenario has changed. This general purpose programming language quickly gained popularity, mostly because of its platform-independent byte-code interpretation. Furthermore, it's touted as a robust, secure, simple, object-oriented, multi-threaded language. All of these features make it very attractive as a database development tool.

To help promote Java usage in database applications, JavaSoft released an API called JDBC(Java Database Connectivity). JDBC is the mechanism for Java applications to talk to a variety of different databases. JDBC creates a programming-level interface for communicating with databases in a uniform manner similar in concept to Microsoft's Open Database Connectivity (ODBC) component which has become the standard for personal computers and LANs. The JDBC standard itself is based on the X/Open SQL Call Level Interface, the same basis as that of ODBC. Platform- and DBMS-independent applications can be developed in Java using JDBC. Any application written to the JDBC standard can access any database with a JDBC driver.

1.1 Motivation

The JDBC API, in conjunction with JDBC drivers, provides a layer of abstraction so that Java programmers can program to a single database API for all databases with relevant drivers.

But, JDBC is not a client server protocol nor does it specify one, to use databases on remote servers. It does not assume anything about the status of the database, be it remote or local. Communication with database is hidden from the programmer and it is entirely implemented in the driver.

Thus, handling remote databases is completely a problem of driver developer i.e.

one has to implement one's own protocol to achieve this. This makes the implementation of JDBC drivers complicated. Maybe this is the reason why majority of JDBC drivers available on the market do not have this capability. However, in case of databases like that of Oracle, developer may wish to use DBMS-specific proprietary protocol like SQL-Net, in his/her driver implementation. But, use of this driver requires that DBMS client software be loaded on the client. Without this arrangement, one cannot write Java applications which make use of remote databases.

Furthermore, Java applications may make use ODBC drivers through JDBC-ODBC bridge. Many desktop databases, including MS Access, Paradox and dBase can only be used in Java applications only through this bridge. But, this bridge is intended only for local databases. So, there is no way to specify a URL for a DSN residing on a remote host.

These problems necessitate the design of a tool which makes driver design simpler by relieving developers of having to worry about remote database access, a tool which can be attached to existing drivers which don't have this capability. And this is the raison-d'etre of this project.

1.2 Objective of the Project

Under this project, we aimed to design and implement a driver which deploys a client/server protocol, and through which one can work on any remote database, having either ODBC driver or JDBC driver which doesn't handle the remote databases. We called it "Meta-driver". Because, it is not associated to any particular DBMS and it acts on top existing drivers.

Further, Meta-driver should act transparently to user .i.e. user interface should not change. Users write their programs according to JDBC specification. They should be able to continue to do so when the Meta-driver is introduced. This boils down to providing driver-like interface on the client side. User first registers this Meta-driver in Driver Manager(like any other driver) and then simply connects to this, specifying the remote host and database that lies on that host.

Preferably, this should eliminate the need to load the binary code on the client,

which is necessary when using ODBC drivers. These ODBC drivers would be loaded on the remote host. So, Meta-driver should make use of those drivers present on the remote system while accessing the databases on it i.e there should be no need to install multiple drivers on the client side to access different databases.

As this acts as a layer on top of drivers, it should not slow down the applications very much. At the same time, it should not introduce any overheads like installing binary code on the client side, configuring clients for specific databases. However, adding this additional layer has some advantages also. At this layer, one can introduce security constraints, requiring additional authentication and validation of client requests. This might be especially important if that database has limited security features.

In short, Meta-driver provides location-transparency, letting users continue to write applications in full JDBC. This helps users to use remote databases which have only ODBC drivers. This can be a very powerful tool on an intranet where data is stored in a different database servers.

1.3 Organization of the Report

The rest of the thesis report is organized as follows. Chapter 2 covers JDBC and ODBC succinctly, the concepts that are required to understand this thesis. Chapter 3 narrates the design of the Meta-driver, its rationale and other rejected design paradigms. Thereafter, Chapter 4 includes the comprehensive implementation details, instructions to install the software, etc. Chapter 5 is an epilogue, wherein conclusion and future work that can be carried out are discussed.

Chapter 2

ODBC/JDBC

In this chapter, the concepts that are related to this project are explicated. These are the topics that are required to be understood before proceeding into design and implementation details of this project. These mainly include ODBC and JDBC.

2.1 Open Database Connectivity

In an effort to standardize an interface to DBMSs, Microsoft created ODBC, based on X/Open definition of SQL CLI(Call Level Interface). ODBC is an API in which application developers can code their programs using ODBC function calls, and each DBMS vendor can provide an ODBC driver for their specific DBMS. An application written for the ODBC API can be used to access any DBMS, given the appropriate ODBC drivers.

2.1.1 Many Vendors - One ODBC Solution

ODBC alleviates the need for independent software vendors and corporate developers to learn multiple APIs. ODBC now provides a universal data access interface. Application developers can allow an application to concurrently access, view, and modify data from multiple, diverse databases.

ODBC is a specification to which developers write an ODBC enabled "front-end"

or "client" desktop application, also known as "ODBC client". This is the application that the computer user sees on the screen.

ODBC architecture comprises the following four layers.

i. ODBC Application

ii. Driver Manager

iii. Driver

iv. DBMS

First one is the ODBC enabled front-end(also called ODBC client). This is an application written to ODBC specification. This can be written in procedural languages like C.

The ODBC Driver Manager is the management layer of ODBC, working between the user and the drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. In addition, the Driver Manager class attends to things like driver login time limits and the printing of log and tracing messages.

Next layer is ODBC driver. This resides between ODBC driver manager and the DBMS being accessed. This is loaded on the front-end computer. ODBC drivers require native code. Basically, ODBC is an API developed specifically for Windows, and most of the drivers are written in C or C++ for Windows or Win32. While some drivers have been ported to Unix or Mac, their availability on other platforms is sporadic at best.

Final layer is "back-end" or "server DBMS". This is the DBMS which resides on a computer that is used to store data, and which allows several users to access data on it. This server application is usually more robust (faster, with centralized security and backups of data, and so forth) than the client application.

Furthermore, using ODBC means reading local configuration settings. ODBC DSN (Data Source Name) identify the different ODBC databases available on a machine, and the drivers they require. DSNs must be configured (in the ODBC control panel on Windows) before any program can use a database through ODBC.

2.2 Need for JDBC

At this point, Microsoft's ODBC (Open DataBase Connectivity) API is probably the most widely used programming interface for accessing relational databases. It offers the ability to connect to almost all databases on almost all platforms. So one may think why not just use ODBC from Java?

The answer is that you can use ODBC from Java, but this is best done with the help of JDBC in the form of the JDBC-ODBC Bridge. The question now becomes, "Why do you need JDBC?" There are several answers to this question:

- ODBC is not appropriate for direct use from Java because it uses a C interface. Calls from Java to native C code have a number of drawbacks in the security, implementation, robustness, and automatic portability of applications.

- A literal translation of the ODBC C API into a Java API would not be desirable. For example, Java has no pointers, and ODBC makes copious use of them, including the notoriously error-prone generic pointer "void *". One can think of JDBC as ODBC translated into an object-oriented interface that is natural for Java programmers.

- ODBC is hard to learn. It mixes simple and advanced features together, and it has complex options even for simple queries. JDBC, on the other hand, was designed to keep simple things simple while allowing more advanced capabilities where required.

- A Java API like JDBC is needed in order to enable a "pure Java" solution. When ODBC is used, the ODBC driver manager and drivers must be manually installed on every client machine. When the JDBC driver is written completely in Java, however, JDBC code is automatically installable, portable, and secure on all Java platforms from network computers to mainframes.

In brief, the JDBC API is a natural Java interface to the basic SQL abstractions and concepts. It builds on ODBC rather than starting from scratch, so programmers familiar with ODBC will find it very easy to learn JDBC. JDBC retains the basic

design features of ODBC; in fact, both interfaces are based on the X/Open SQL CLI (Call Level Interface). The big difference is that JDBC builds on and reinforces the style and virtues of Java, and, of course, it is easy to use.

More recently, Microsoft has introduced new APIs beyond ODBC: RDO, ADO, DAO, and OLE DB. These designs move in the same direction as JDBC in many ways, for example, in being object-oriented interfaces to databases based on classes that can be implemented on ODBC. However, we did not see functionality in any of these interfaces compelling enough to make them an alternative basis to ODBC, especially with the ODBC driver market well-established. Mostly they represent a thin veneer on ODBC.

2.3 Java Database Connectivity

The Java Database Connectivity Standard (JDBC) is part of the Java Enterprise API. JDBC is an SQL based database access interface. It provides Java Programmers with a uniform interface to a wide range of relational databases, and also provides a common base on which higher level tools and interfaces can be built.

The JDBC API defines classes to represent constructs such as database connections, SQL statements, result sets, and database metadata. JDBC allows a Java-powered program to issue SQL statements and process the results.

In conjunction with JDBC, JavaSoft has released an JDBC-ODBC bridge implementation that allows any of the dozens of available ODBC drivers to operate as JDBC drivers.

2.3.1 Outline of JDBC functionality

JDBC itself, like most driver specifications, is pretty low-level and functional. Here in a nutshell is a short review of how JDBC works.

JDBC uses a simple class hierarchy for database objects. The classes are contained in the java.sql.* package (which were released after JDK 1.0, but will be included in JDK 1.1). JDBC itself is just a specification; literally, the java.sql.* classes are

descriptions of classes and methods that must be written in order to produce a JDBC driver. The functionality of JDBC is described here in terms of the normal progression through a database session, from connecting to disconnecting.

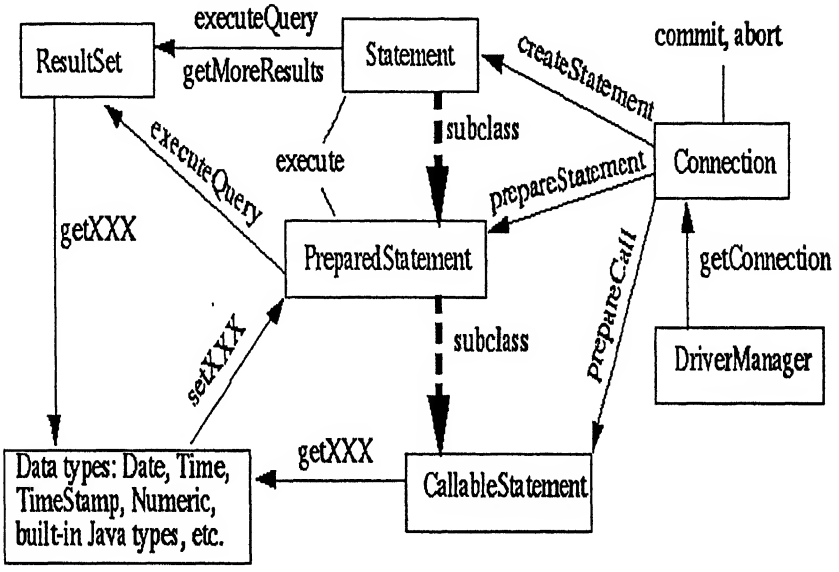


Figure 1: Typical JDBC Scenario

Obtaining a Connection

There are three classes that pertain to opening a connection to the DBMS: *java.sql.DriverMan* *java.sql.Connection*, and *java.sql.DatabaseMetaData*. The DriverManager loads the appropriate JDBC driver, after which you can create Connection object. The Connection object is very important in JDBC; many other objects are constructed and many methods are executed in the context of a *java.sql.Connection*. The DatabaseMetaData returns information about the client's connection and interesting information about the database to which the client has connected.

▮ *Making a query and getting its results*

After connecting, most clients will perform a query of some sort, either a select statement, or an insert, update, or delete statement. The *java.sql.Statement* class is used to compose and execute particular kinds of SQL queries on the DBMS. The results of a query are used to create a *java.sql.ResultSet*. The JDBC ResultSet is navigable in only one direction – next – and there are limitations on manipulating results. Although *ResultSet* doesn't allow for very sophisticated query management, it is a good foundation upon which to build other higher-level APIs with elegant, easy-to-use objects for data management. Meta information about the *ResultSet* itself is contained in the *ResultSetMetaData* objects.

You can also execute stored procedures on a database with two subclasses of *java.sql.Statement*, *java.sql.PreparedStatement*, and *java.sql.CallableStatement*.

▮ *Other JDBC classes*

Other JDBC classes are supplied for utility purposes, like *java.sql.Types*, that encapsulate Java types for database use, and *java.sql.Date*, *java.sql.Time*, and *java.sql.Timestamp*. There are some classes that a developer writing an application may never use directly, but which the vendor who implements the JDBC specification will use internally, like *java.sql.DataTruncation*, *java.sql.DriverInfo* and *java.sql.DriverPropertyInfo*. Exception handling is provided for in the classes *java.sql.Exception* and *java.sql.Warning*.

In addition to the classes prescribed by the JDBC specification, a JDBC implementation may also write certain extensions to JDBC that fulfill operations specific to a particular DBMS. For example, WebLogic's jdbcKona/Oracle driver provides an extension to JDBC for creating and using Oracle sequences.

2.3.2 JDBC Drivers

Chiefly, the idea of JDBC is to separate out into one code layer all the code that is common to accessing any SQL-2 database. Doing this requires, then, another layer that contains the code specific to accessing the specific database. This code-specific

layer is called "JDBC Driver".

Based on their implementation, JDBC drivers fall into the following taxonomy.

I. *JDBC-ODBC Bridge Driver:*

This is intended to provide database access to JDBC applications through existing ODBC drivers. This bridge translates JDBC calls to ODBC calls on the client machine. To ODBC, it appears as normal application.

To make use of this driver, ODBC binary code must be loaded on each client machine. As a result, this kind of driver is appropriate on a corporate network where client installations are not a major problem, or application server code written in Java in a three-tier architecture.

JavaSoft has provided this bridge as the sun.jdbc.odbc Java package and contains a native library used to access ODBC. Presumably, Microsoft's JDBC-ADO driver would also fall into this category as it maps JDBC to another database API used to access a broad variety of databases.

II. *Native-API partly-Java Driver:*

This is Type 2 driver. This kind of driver converts JDBC calls into calls on the client API for Oracle, Sybase or other DBMS. Like bridge driver, this also requires that some binary code be loaded on each client machine.

III. *Net Protocol All-Java Driver:*

Other names for this driver are Type 3 driver and JDBC-Net pure Java driver.

This driver translates JDBC calls into a DBMS-independent net protocol which is then translated to a DBMS protocol by a server. This net server middleware is able to connect its pure Java clients to many different databases. The specific protocol used depends on the vendor.

In general, this is the most flexible JDBC alternative. It is likely that all vendors of this solution will provide products suitable for intranet use. In order for these products to also support Internet access, they must handle the additional requirements for security, access through firewalls, and so forth, that the Web imposes.

IV. *Native Protocol Pure-Java Driver:*

This type of driver, written entirely in Java, speaks directly to the database server using the database server's native network protocol. This allows a direct call from the client machine to the DBMS server and is a practical solution for Intranet access.

Since many of these protocols (such as SQL-Net in case of Oracle) are proprietary, this type of drivers are being developed by DBMS vendors only.

The expectation is that eventually driver categories 3 and 4 will be the preferred way to access databases from JDBC. Driver categories 1 and 2 are interim solutions where direct pure Java drivers are not yet available. There are possible variations on categories 1 and 2 that require a connector, but these are generally less desirable solutions. Categories 3 and 4 offer all the advantages of Java, including automatic installation (for example, downloading the JDBC driver with an applet that uses it).

2.3.3 Performance of Drivers

The performance of JDBC drivers depend on several factors. Of them, the following ones can be considered important.

a. the quality of the driver code

b. the size of the driver code

c. the database server and its load

d. network topology

e. the number of times your request is translated to a different APIs.

These factors vary depending on the type of the driver. Consider JDBC-ODBC bridge. Its performance depends on the co-operation of several different components such as JDBC driver manager, ODBC driver manager, ODBC driver and the database. So, we can conclude that this is only as strong as the weakest of the links in the chain that support it. Of all types of drivers, Native Protocol Pure Java Driver can be considered fastest.

2.4 JDBC versus CGI

Java and JDBC applets are more powerful, offer lighter server loads, lighter network loads and less development time than cgi scripts and programs.

CGI scripts and programs are sessionless and restricted to using HTML as a user interface, making it very difficult to track and automatically respond to user actions. Overcoming these limitations is very difficult, requiring valuable developer time to architect a system that does not appear session-less. In addition, HTML's limited number of user interface components means that your custom applications will often have to be re-designed to fit the Net.

Java has overcome both of these issues. Because Java is not session-less, an applet can easily keep track of what the user has done and react to user events accordingly. And Java allows full user interface design, giving developers complete control over how their programs appear and perform.

Because CGI scripts and programs must execute on the server, server loads are much higher. An Internet server utilizing CGI scripts and programs can easily spend 80the script/program. Needless to say, server performance decreases with the complexity of the script. As a result, servers employing a JDBC solution that can handle many more users/hour than with CGI scripts. The Simba JDBC solution removes the load from the server and distributes it to the client machines, effectively sharing the load on the server. The load on the client machine is still very small, however, and does not require a lot of processing power.

Because of HTML's inability to react to user input, CGI scripts are used to provide interaction. Typically, a user is asked to input a small amount of information, and is returned a large HTML page, generating a high network load. With Java, only the input/changed information is transmitted between the server and the applet, decreasing network transmissions of 20 kilobytes (with the CGI script) to under 1 kilobyte.

Custom CGI scripts and programs are created for a particular purpose in whatever language the programmer feels comfortable with (most commonly C or Perl). There

are no standards governing their creation, so they often require huge amounts of documentation. Even C++ doesn't force developers to write OO code, possibly resulting in code maintenance as high as that for C or PERL. Because of this, developers can't rely on third-party products to supply them with appropriate code modules. Java, on the other hand, is a standard language that is object-oriented and componentized by nature. An almost unlimited amount of third-party reusable components contribute to drive down development time, and reduce code maintenance and documentation. The powerful tools available to object-oriented developers ease code modularization, making large projects easier to understand.

2.5 JDBC versus ODBC

With the Internet and Java taking the technological world by storm, many people are wondering what to make of all the new standards. The database world is no exception, with two main standards currently vying for acceptance. Open DataBase Connectivity (ODBC) is a powerful standard co-developed by Microsoft and Simba Technologies to create a common interface for relational databases. The second standard, introduced by Sun Microsystems, is Java DataBase Connectivity, or JDBC. As has been discussed earlier, it is similar to ODBC, but JDBC is an object-oriented interface based on the cross-platform Java development language. For IT managers and application developers, deciding which standard to support with their products has become very confusing. Anyway, the answer is not so easy.

Consider speed. ODBC implementations are currently faster than that of JDBC ones because Java virtual machines tend to run the applications slower than native machine code does. So, if speed is of prime importance, one can go for ODBC. But then, he has to use procedural languages like C in their applications (Here, JDBC-ODBC bridge would not be useful as it slows down the applications).

Also, most desktop query tools with which end-users are familiar are ODBC based. Thus, it enjoys greater installed user base and companies need not spend money on re-training.

Moreover, ODBC is a more stable standard as it has been around much longer

than JDBC. But, Java and JDBC are emerging technologies and as such, are dynamic, flexible and future-oriented. JDBC outshines ODBC at the following issues.

- *Low memory requirements for drivers:*

  JDBC drivers take less memory when compared with ODBC drivers. So, in case of any software running on a low memory client, it would be better to employ JDBC as standard.

- *Platform independence:*

  Java is a write-one, run-anywhere environment, scalable from smart cards up to enterprise servers. Organizations that are unsure of their main platform, or companies that require support for multiple platforms, should consider the JDBC standard.

One more benefit of using JDBC is, there are no dependencies on third parties. Because of Java's open architecture, platform support is not an issue. You will have no dependencies on software vendors to support your platform, and so no need to port your software between platforms.

Clearly, any new applications being developed should take into account the JDBC standard. Ideally, developers should design database drivers for both standards, with ODBC being given priority because JDBC connectivity can be obtained from an ODBC interface anyway.

Chapter 3

Meta-driver Design

This chapter elucidates the design aspects of the meta-driver and its structure. To start with, it describes the design goals, which are to be kept in mind while designing and implementing. Next, an elaborate design of the Meta-driver follows. This includes the architecture of Meta-driver and a detailed discussion of what changes it brings into the current system. Thereafter, a detailed discussion follows, regarding the the possible approaches of solution to the Meta-driver, which protocol is to be deployed and the rationale behind its choice, etc.

3.1 Design Goals

Primarily, this software should be designed as a separate component and should not be tied down to any platform or DBMS. This also means, it should allow user to attach this Meta-driver to any existing JDBC/ODBC drivers with possible minimal effort . This should be implemented in such a way that one can easily embed this into a new driver (either for existing DBMS or for a new DBMS) being designed. That should relieve the developer of having to bother about handling remote databases facility.

Besides, it should be transparent to users and is not expected to change their interface (more accurately, Application Programming Interface, API). This API happens to be JDBC in this case. This means, the programmer need not learn a new

API to use Meta-driver. They would continue to use the same as they did before, but instead of connecting to some JDBC/ODBC driver, they would then connect to Meta-driver and work.

Another important design goal is, it should require minimal or no client configuration. Otherwise, users would be burdened with this additional configuration, along with initial settings required by ODBC drivers. Moreover, it should not require loading of binary code on clients. Otherwise, users would not be able to use the Meta-driver in applets, as the applet security model prevents loading untrusted native code over the network.

Apart from all these, its performance should be reasonable. As it adds one more layer to the system, it should not introduce much maintenance overhead and should not slow down the applications much.

3.2 Position of Meta-driver in the System

Logically, Meta-driver is positioned between the application program layer and the layer containing JDBC/ODBC drivers in the system. Mostly, in practice, application program resides on client machine, and drivers reside on remote machine along with database to be accessed.

One interesting side-effect that meta-driver brings into the system is, it converts the two-tier architecture to three-tier. The normal two-tier scenario is like as follows. Application program, JDBC/ODBC drivers constitute one first tier and database, the second one. The Java applet or application talks directly to database. When meta-driver is deployed, it introduces a third layer in between these two, thereby converting the system to three-tier. Sometimes, this middle layer is also known by the name "middleware".

3.2.1 Problems with Two-Tier Architecture

Firstly, it requires that proprietary ODBC and JDBC drivers be deployed and

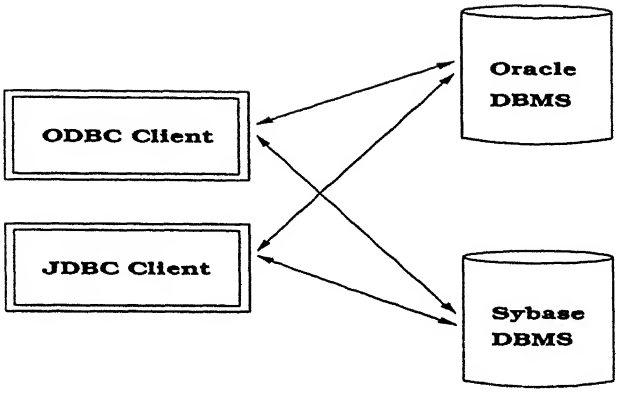


Figure 2: Two-tier Architecture

maintained on each desktop, for each database which the organization wishes to provide users access to. This means, if there are twenty DBMSs, each client machine should be loaded with twenty drivers. As the number of clients increases, administration would become harder.

Secondly, this is less flexible and harder to maintain. Suppose, the administration wants to replace some driver X by driver Y. Then, on each client machine, this modification should be performed. Same situation arises when one wants to add a new driver, because some DBMS is introduced. Even though this problem may seem less severe for smaller organizations, but surely is more severe in case of large number of client machines.

This also means drastic increase in user support, maintenance and data source administration. Each client machine should be loaded with binary code associated with drivers and users have to administer the data sources on their own. So, every time a new database is created, though centrally, every user has to configure this information , using data source administrator on his/her machine. This problem is especially evident in case of users having to access through ODBC drivers.

This two-tier system introduces other administration headaches also. Each users have to change their configuration when there are changes to central databases like change of names of databases, deletion of some database, addition of new database. Apart from all these, it would be also difficult to impose security constraints on users.

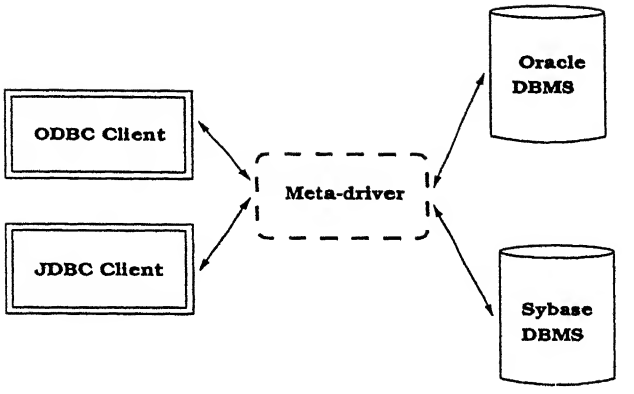


Figure 3: Three-tier Architecture

The ideal solution to the above mentioned problems is to make the system three-tier. This is what exactly the meta-driver does. In a three-tier scenario, Java applications interact with a "middle tier" of services on the network, which in turn access the DBMS through some client/server protocol like RPC (remote procedure call) or an ORB (object request broker). Three-tier services could be more secure for restricted data banks, since all data queries are funneled through one central "gateway".

3.3 Proposed Solution: Meta-driver

The Meta-driver requires implementing a JDBC driver, employing some client/server protocol, and establishing a server which handles connections from the clients and interacts with the databases. We named it so, because this driver does not interact with any particular DBMS directly, but only with other drivers.

Meta-driver is chiefly made up of two components. They are

i. MetaDriver Server component:

   This is the component that resides on server side, where databases and drivers are loaded. This includes a server process running on this machine. This process handles all the requests from clients, sets up connections to databases, etc. This should be able to use any driver that has been registered with the Driver Manager.

ii. MetaDriver Client Component :

Users/Clients are provided with a JDBC driver which communicates to the server. Using this, they can make use of server-side JDBC/ODBC drivers to work with remote databases.

Apart from these two components, there is an important tertiary component. That is the client/server protocol that should be deployed, whereby client drivers interact with databases residing on server machine. This refers to how clients communicate with server process, how objects are passed between client and server, how exceptions are handled etc.

3.3.1 Choice of Protocol

The choice of protocol is an important design consideration. It determines some of the major implementation details, for example, whether the driver has to be a *"pure java"* or *"partly java"* driver.

Partly java approach, whereunder we might implement our own client/ server protocol using either *C Socket interface* or *Sun RPC*. In this case, we need to integrate C code with java using native methods (or Java Native Interface(JNI), which has been standardized recently). This may also cause problems in portability of the code, across platforms. This requires some binary code be loaded on clients, which conflicts with our design goals. In addition, as these are not object-oriented, handling objects is not direct and complex. Apart from all these, native methods in Java are not secure also. Due to these problems, we avoided this approach.

Consequently, *pure Java* approach was considered. That constrains complete Meta-driver implementation be done in java. In this case, no loading of binaries is required on clients. Under this approach, the hitherto available options are

a. *Java Socket Interface*

b. *Java Remote Method Invocation (RMI)*

c. *Java IDL (CORBA)*

For a basic communication mechanism, the Java language supports sockets, which are flexible and sufficient for general communication. However, this is a very low-level API and it requires the client and server to engage in applications-level protocols to encode and decode messages for exchange, and the design of such protocols is cumbersome and can be error-prone. So, we decided not to deploy Java Sockets, because of above mentioned reasons.

An alternative to sockets is Remote Procedure Call (RPC), which abstracts the communication interface to the level of a procedure call. Instead of working directly with sockets, the programmer has the illusion of calling a local procedure, when in fact the arguments of the call are packaged up and shipped off to the remote target of the call. RPC systems encode arguments and return values using an external data representation, such as XDR.

RPC, however, does not translate well into this project, where communication between program-level objects residing in different address spaces is needed. In order to match the semantics of object invocation, distributed object systems require remote method invocation or RMI. In such systems, a local surrogate (stub) object manages the invocation on a remote object.

Subsequently, use of RMI and Java IDL were contemplated. RMI enables the programmer to create distributed Java-to-Java applications. A Java program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in the bootstrap-naming service provided by RMI, or by receiving the reference as an argument or a return value. RMI uses Object Serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism.

Java IDL provides the necessary means to allow Java applications to manipulate CORBA objects on remote servers as if they were local Java objects. Many Java IDL compilers implement only a part of the CORBA standard, but they are the only way to functional Java/CORBA solution.

The following facts were considered to decide upon the protocol.

* RMI is an API standard for building distributed Java systems and CORBA, for

building heterogeneous distributed systems. That means, in case of RMI, both client and server should be written in Java. But, in case of CORBA, there are no such constraints. Client may be in Smalltalk and server, in java. Note that our project needs to be implemented in java completely.

* By using RMI, one gains the benefits of full java semantics. But, CORBA is not tied down by any one particular language and so offers no such benefits. So, while working with RMI objects, one need not take care of security issues separately, as Java provides all of them.

* RMI incorporates distributed garbage collection whereas CORBA does not. But, RMI is not capable of handling massive distributed systems whereas CORBA is meant for that.

* Finally, Java IDL compilers were in alpha stages when this project had started (for that matter, even now, there are hardly any commercial compilers, to the best of our knowledge).

Keeping in view the project requirements, we have chosen to deploy RMI in the implementation of Meta-driver. This does not need loading of new software on the systems, but only JDK1.1 compiler. (Had we chosen Java IDL, it would have required installing of Java IDL compiler separately.)

To sum up, Meta-driver was designed to serve as additional layer between client and databases. It is only client component of Meta-driver that needs to be loaded on client machines. All other ODBC/JDBC drivers will be loaded on remote machine, where server component of Meta-driver runs. It was decided to implement this in Java completely, to avoid loading of binaries on clients. And, client part of Meta-driver communicates with the server using RMI.

Chapter 4

Meta-driver Implementation

This chapter illustrates the implementation details of this project. Initially, it describes the architecture of the Meta-driver, its components, and how they were implemented, etc. The following section incorporates detailed instructions required for the installation and configuration of Meta-driver software. Subsequently, a section follows containing the brief description of all the classes in this project and their intended functions. Finally, a sample client program is explained, which executes SQL statements on a remote server.

4.1 Architecture

Meta-driver is a client/server JDBC Driver that relies on Java RMI. All JDBC classes (like Connection, ResultSet, etc...) are distributed as RMI objects, so that one can distribute as one likes the access to any database supporting the JDBC API.

In other words, Meta-driver is a bridge to allow remote access to JDBC drivers.

Suppose one implements a JDBC Driver: with Meta-driver, he/she just has to implement the JDBC classes locally, not bothering with remote access.

Meta-driver was implemented as one Java package, so that one can easily include in their code to make use of this software. This also helps attach the Meta-driver to newly designed drivers.

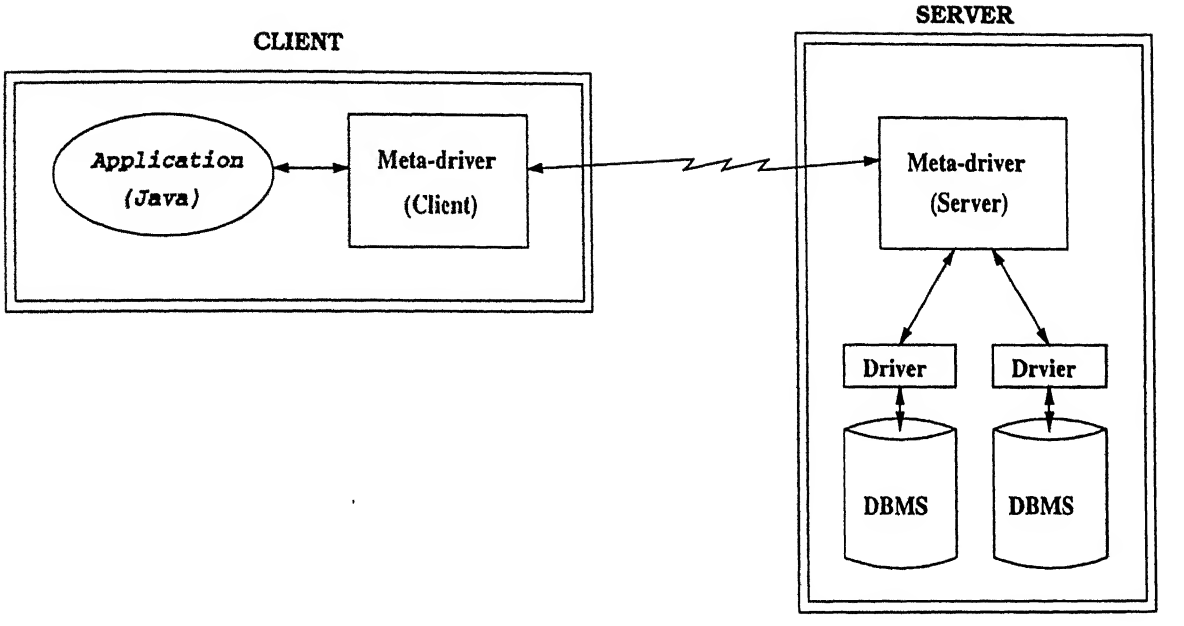


Figure 4: Meta-driver Architecture

I. Meta-driver Server component:

On the server side, a daemon (*RemoteServer* class) runs: it offers an RMI server object that implements the whole interface of a JDBC Driver object, and can give access to any JDBC/ODBC Driver registered in the JDBC DriverManager on the server side. This is like an intermediate server to mediate between clients and the DBMS.

The main functions of this server include

a. allowing clients to make connections to the server machine,

b. interacting with DBMS through drivers,

c. sending results back to the clients,

d. handling exceptions (if any).

II. Meta-driver Client-side JDBC Driver:

This is the part of the Meta-driver which is to be loaded on all the client machines. Note that, earlier, all client machines were to be loaded with all the

JDBC/ODBC drivers to access respective databases and now, it is sufficient to load just the Meta-driver client and all the drivers can be loaded on one centralized machine(server). The Meta-driver JDBC Driver runs on the client side: the JDBC classes it implements contact the Meta-driver Server to access databases.

4.2 Installation and Configuration

The steps to install the Meta-driver in the system are given hereunder.

i. Make the CLASSPATH variable point on the MetaDriver.jar package

```
e.g. SET CLASSPATH=%CLASSPATH%;C:\Kataru\Thesis\Test\MetaDriver;
```

ii. Start RMI registry.

The RMI registry is a simple server-side bootstrap name server that allows remote clients to get a reference to a remote object. It is typically used only to locate the first remote object an application needs to talk to. That object in turn will provide application specific support for finding other objects.

To start the registry on the server, execute the `rmiregistry` command. This command produces no output and is typically run in the background. For example, on Windows 95 or Windows NT:

```
start rmiregistry (Use 'javaw' if 'start' is not available.)
```

And on Solaris:

```
rmiregistry &
```


The registry by default runs on port 1099.

One must stop and restart the registry any time one modifies a remote interface or use modified/additional remote interfaces in a remote object implementation. Otherwise, the class bound in the registry will not match the modified class.

iii. Start the Meta-driver Server component:

* `java MetaDriver.RemoteServer [driverList]` where `driverList` is a list of JDBC Driver classes available on the server (e.g. jdbc.odbc.JdbcOdbcDrive
* One can also declare one's driver list in the jdbc.drivers System property (java -Djdbc.drivers=driverList MetaDriver.RJJdbcServer) Then, a remote JDBC client can access any local database.

4.3 Overview of the Classes

In this section, brief descriptions are given regarding all important classes used in this project.

- `public class MetaConn`
  `extends Object`
  `implements Connection`

A Connection represents a session with a specific database. Within the context of a Connection, SQL statements are executed and results are returned.

A Connection's database is able to provide information describing its tables, its supported SQL grammar, its stored procedures, the capabilities of this connection, etc. This information is obtained with the getMetaData method.

Note: By default the Connection automatically commits changes after executing each statement. If auto commit has been disabled, an explicit commit must be done or database changes will not be saved.

- `public class MetaStmt`
  `extends Object`
  `implements Statement`

A Statement object is used for executing a static SQL statement and obtaining the results produced by it.

Only one ResultSet per Statement can be open at any point in time. Therefore, if the reading of one ResultSet is interleaved with the reading of another, each must have been generated by different Statements. All statement execute methods implicitly close a statement's current ResultSet if an open one exists.

- `public class MetaResultSet`
  `extends Object`
  `implements ResultSet`

A ResultSet provides access to a table of data generated by executing a Statement. The table rows are retrieved in sequence. Within a row its column values can be accessed in any order.

A ResultSet maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The 'next' method moves the cursor to the next row.

The getXXX methods retrieve column values for the current row. One can retrieve values either using the index number of the column, or by using the name of the column. In general using the column index will be more efficient. Columns are numbered from 1.

For maximum portability, ResultSet columns within each row should be read in left-to-right order and each column should be read only once.

For the getXXX methods, the JDBC driver attempts to convert the underlying data to the specified Java type and returns a suitable Java value. See the JDBC specification for allowable mappings from SQL types to Java types with the ResultSet.getXXX methods.

Column names used as input to getXXX methods are case insensitive. When performing a getXXX using a column name, if several columns have the same name, then the value of the first matching column will be returned.

A ResultSet is automatically closed by the Statement that generated it when that Statement is closed, re-executed, or is used to retrieve the next result from a sequence of multiple results.

The number, types and properties of a ResultSet's columns are provided by the ResultSetMetaData object returned by the getMetaData method.

- `public class MetaClientDriver`
  `extends Object`
  `implements Driver`

The Java SQL framework allows for multiple database drivers.

Each driver should supply a class that implements the Driver interface.

The DriverManager will try to load as many drivers as it can find and then for any given connection request, it will ask each driver in turn to try to connect to the target URL.

It is strongly recommended that each Driver class should be small and stand-alone so that the Driver class can be loaded and queried without bringing in vast quantities of supporting code.

When a Driver class is loaded, it should create an instance of itself and register it with the DriverManager. This means that a user can load and register a driver by doing Class.forName("foo.bah.Driver").

- `public class MetaDbMData`
  `extends Object`
  `implements DatabaseMetaData`

This class provides information about the database as a whole.

Many of the methods here return lists of information in ResultSets. One can use the normal ResultSet methods such as getString and getInt to retrieve the data from these ResultSets. If a given form of metadata is not available, these methods should throw a SQLException.

Some of these methods take arguments that are String patterns. These arguments all have names such as fooPattern. Within a pattern String, "match any one character. Only metadata entries matching the search pattern are returned. If a search pattern

argument is set to a null ref, it means that argument's criteria should be dropped from the search.

A SQLException will be thrown if a driver does not support a meta data method. In the case of methods that return a ResultSet, either a ResultSet (which may be empty) is returned or a SQLException is thrown.

- `public class MetaRSMData`
  `extends Object`
  `implements ResultSetMetaData`

A ResultSetMetaData object can be used to find out about the types and properties of the columns in a ResultSet.This class allows the application developer access to metadata like `ColumnName`, `ColumnType`, `getColumnTypeName`.

- `public class MetaPrepStmt`
  `implements PreparedStatement`

A SQL statement is pre-compiled and stored in a PreparedStatement object. This object can then be used to efficiently execute this statement multiple times.

Note: The setXXX methods for setting IN parameter values must specify types that are compatible with the defined SQL type of the input parameter. For instance, if the IN parameter has SQL type Integer then setInt should be used.

If arbitrary parameter type conversions are required then the setObject method should be used with a target SQL type.

- `public class MetaCallStmt`
  `implements CallableStatement`

CallableStatement is used to execute SQL stored procedures.

JDBC provides a stored procedure SQL escape that allows stored procedures to be called in a standard way for all RDBMS's. This escape syntax has one form that includes a result parameter and one that does not. If used, the result parameter

must be registered as an OUT parameter. The other parameters may be used for input, output or both. Parameters are referred to sequentially, by number. The first parameter is 1.

```
{?= call [ , , ...]}
{call [ , , ...]}
```

IN parameter values are set using the set methods inherited from PreparedStatement. The type of all OUT parameters must be registered prior to executing the stored procedure; their values are retrieved after execution via the get methods provided here.

A Callable statement may return a ResultSet or multiple ResultSets. Multiple ResultSets are handled using operations inherited from Statement.

For maximum portability, a call's ResultSets and update counts should be processed prior to getting the values of output parameters.

4.4 An Example Client

Here, a simple Java client which uses Meta-driver to access database on a remote machine, is explained. The client machine does not contain any drivers but Meta-driver client component. And the destination machine is loaded with the necessary drivers and DBMSs.

```
import java.io.*;
import java.sql.*;
import MetaDriver.*;
import java.rmi.*;
import java.net.*;
```

First, all the packages that are needed to make this program run were imported. Meta-driver was also used and it was included by importing *MetaDriver* package.

```
// ClientSample is a class, which acts as a client to the Meta-driver
// and executes SQL statements on the remote database

class ClientSample{


public static void main(String args[]){

try{

  MetaDriver.DbUrlLib.registerMetaDriver("pc82");
  String sampleurl = MetaDriver.DbUrlLib.getTargetUrl
           ("jdbc:odbc:acadinfo.mdb");
```

The next part is configuring the Meta-driver and providing it with details of the destination machine and the database to which access is needed. For this purpose, the class *DbUrlLib* was used, which configured the Meta-driver by registering Meta-driver with the Driver Manager in the system. Then, user supplied the remote machine name through *registerMetaDriver* function. And the database URL, through *getTargetUrl* function. This function returned a string *smapleurl* which would be used as handle to that database, in subsequent sections of the program. Here, the target machine is *pc82* and the database is that of *MSAccess*.

```
// Here onwards, programmer can make use of full JDBC functionality
// to write the applications

Connection con = DriverManager.getConnection(sampleurl);

Statement stmt = con.createStatement();

String query1 = "SELECT * from deptinfo;";
```

```
ResultSet rset = stmt.executeQuery(query1);

while(rset.next()){

ResultSetMetaData rsmd = rset.getMetaData();

for(int i=1;i<=rsmd.getColumnCount();i++){
System.out.print(rset.getString(i)+ ",");
}
System.out.println("");
}

rset.close();
stmt.close();
con.close();
}
```

This part is almost self-explanatory. First, a connection was made to the database using the *sampleurl* returned by *getTargetUrl* function.Each connection to each database can issue any number of SQL queries, each returning one *ResultSet*. Likewise, after establishing the connection, a statement object was created from it. This was used in executing the required SQL statement on that database. The statement was executed by *executeQuery* function, and the result of the execution were received in a *ResultSet* object. Through *while* loop, these results were extracted row-by-row, and printed.

```
catch(SQLException e){

// print the SQL State from java.sql.SQLException
   System.out.println("SQL State: "+e.getSQLState());

     // print the Error Code from java.sql.SQLException
       System.out.println("Error Code: "+e.getErrorCode());
```

```
// use the superclass (java.lang.Throwable) method
   // to get the exception's message
System.out.println("Message: "+e.getMessage());

        // use the superclass (java.lang.Throwable) method
// to print the stack trace
  e.printStackTrace();
  }
} /* End of main */
} /* End of the class */
```

This is exception handling part. While these examples merely print the messages to the screen, other statements to work around the exception could be written into the `catch` block. An example would be rolling back the database table in the event of a fatal error.

Chapter 5

Conclusion and Future Work

As offices continue to automate their business, the need to organize, maintain, and access electronic data increases. Database vendors not only have to provide a means to data access, but they must also ensure it regardless of the platform, DBMS which the data is stored on. ODBC and JDBC have been developed to address this problem. But, they have left the problem of accessing remote databases to the driver designers.

A generic JDBC Meta-driver was designed and implemented using RMI to enable the users to use existing JDBC/ODBC drivers (which do not support client/server protocol) to work with remote databases. This also simplified the design of new drivers in that the designer need not care about remote database access and can use this one instead.

Since all JDBC classes were distributed objects, so one can distribute as he likes the access to any database supporting the JDBC API.

Meta-driver was implemented and introduced as a third-tier to the present two-tier system. Because three-tier services could be more secure for restricted data banks, since all data queries are funneled through one central "gateway". For example, a secure intermediate layer can provide means for the developer to shield the client from direct access to DBMS.

This, in one sense, has extended the capabilities of JDBC-ODBC bridge. One can use Meta-driver along with the JDBC/ODBC Bridge on Windows NT to make all desktop databases residing on it remotely accessible in Java .

5.1 Limitations

The Meta-driver was found to have the following limitations at the moment.

- It was observed running more slowly than expected, the reasons being the java interpreter, the higher level protocol used(RMI), and the increased number of layers in program execution.
- Java RMI is an integral part of JDK1.1.x and most of the browsers support JDK1.0.x only. So, Meta-driver in applets may not work with all browsers (neither on all platforms, for that matter). For example, Netscape Navigator 4.03 or later versions only support JDK1.1. In case of MSIE (MicroSoft Internet Explorer), versions earlier than 4.x do not support this.
- Meta-driver applications can connect remotely to only one server (with possibly several databases on it!)
- *getObject()* methods can only return serializable objects (due to RMI distribution). An object is said to be serializable if its class implements either the Serializable or the Externalizable interface. This is the key to storing and retrieving objects and it represents the state of objects in a serialized form sufficient to reconstruct the objects.

5.2 Future Work

The following future work can be undertaken as the extension to this project/software. All these extensions centre around further strengthening of the middle layer.

i. *Addition of some security features to the middle-tier.* This may include maintaining a list of trusted clients at the server end, allowing users to connect to the server through a user-name and a password, etc.

ii. *Optimization of connections the clients open.* At present, if hundred clients open connections to the same database, then there would be hundred connections to

that DBMS. But, both network traffic and DBMS use could be reduced if server opens a small number of connections to the database, leave them open, and then allow (trusted) clients to use connections from the pool.

iii. In the same fashion, one can reduce network traffic also by limiting and managing the query results that are passed to the client i.e. results can be cached on the intermediate server, and then parcel out those results as (and only if) the client asks for them.

iv. *Making the intermediate server as separate proxy.* The web server from which the applet was retrieved serves as a proxy gateway, by which applets can in fact communicate with any remote hosts the proxy allows. The native code library is housed on a second host, and serves as a gateway to a third tier, the DBMS. The database client may instantiate a "session" with the native code library gateway, which in turn connects to the remote database system.

While working out the above mentioned extensions to this project, efficiency and the convenience to the user should be considered prominent.

Bibliography

[1] Brian Jepson, *"Java Database Programming"*
John Wiley and Sons Inc., New York, 1997.

[2] R. Elmasri and S. Navathe, *"Fundamentals of Database Systems"*
The Benjamin/Cummings Publishing Co. Inc., Second Edition, 1994.

[3] M.Morrison et.al., *"Java Unleashed"*
Sams Net, Indiapolis, Indiana. 1st Edition, 1996.

[4] Hamilton, Cattell et. al., *JDBC: Database Access with Java – A tutorial and Annotated Reference*
JavaSoft Press, Addison-Wesley.
ftp://splash.javasoft.com/pub/jdbc.book.examples.zip

[5] Graham Hamilton and Rick Cattell, *"JDBC: A JAva SQL API, Ver. 1.20"*
JavaSoft Inc., January 1997.
ftp://splash.javasoft.com/pub/jdbc-spec-0120.ps

[6] Dan Brookshier, *Java Beans: Developer's Reference*
New Riders Publishing, Inidanapolis, Indiana. 1997.

[7] Ken Arnold and James Gosling, *The Java Programming Language* Addison-Wesley Publishing Co., Inc., New York.

[8] JavaSoft Inc., *"JDBC 1.2 API Documentation"*
ftp://splash.javasoft.com/pub/jdbc-api-1-0120.ps ftp://splash.javasoft.com/pub/jdbc-api-2-0120.ps

[9] Sun MicroSystems, Inc., Oct 1997, *"Java Remote Method Invocation Specification"*
http://java.sun.com/products/rmi/rmi-spec-JDK1.2.ps

[10] Rawn Shah, *"Integrating Databases with Java via JDBC"*
Java World, IDG's Magazine for Java Community, May 1996.
http://www.javaworld.com/javaworld/jw-05-1996/jw-05-step.html

[11] Pratik Patel and Karl Moss, *"Java Database Programming with JDBC"*
Coriolis Publications, 1996.
http://www.coriolis.com/Site/MSIE/Books/Ind/jdpjdbc.html

[12] JavaSoft Inc., *"The JavaSoft's JDBC Site"*
http://splash.javasoft.com/jdbc/index.html

[13] Michael Shoffner, *"Scaling an Application from two-tier to three-tier with JDBC"*
Java World, IDG's Magazine for Java Community, June 1997.
http://www.javaworld.com/javaworld/jw-06-1997/jw-06-step.html

[14] Sun MicroSystems Inc., *"The Official JDBC FAQ"*
http://java.sun.com/products/jdbc/jdbc-frequent.html

[15] Bryan Morgan, *"CORBA meets Java"*
Java World, IDG's Magazine for Java Community, October 1997.
http://www.javaworld.com/javaworld/jw-10-1997/jw-10-corbajava.html

[16] Sander Spruit, *"Reflections on Java, Beans, and Relational Databases"*
Java World, IDG's Magazine for Java Community, September 1997.
http://www.javaworld.com/javaworld/jw-09-1997/jw-09-reflections.html

[17] Weblogic Inc., *"Choosing a JDBC Driver"*
http://www.weblogic.com/

[18] Sun MicroSystems Inc , *"JDBC-ODBC Bridge Driver"*
http://www.javasoft.com/products/jdbc/doc/guide/bridge.doc.html

[19] Openlink Software Inc., *"Openlink JDBC Whitepaper"*
http://www.openlinksw.com/

[20] YoYo Publications, *"JDBC Frequently Asked Questions"*
http://www.yoyoweb.com/Javanese/JDBC/DriverTypes.html

[21] YoYo Publications, *"JDBC Driver Types"*
http://www.yoyoweb.com/Javanese/JDBC/DriverTypes.html

[22] JavaSoft Inc., *"JavaSoft's Drivers Page"*
http://www.javasoft.com/jdbc/jdbc.drivers.html

[23] Brian Jepson, IDS Net Inc., *Free ODBC Page"*
http://users.ids.net/ bjepson/FreeODBC/

[24] George Reese, *"DB Programming with JDBC and Java"*
O'Reilly & Associates, 1997.
http://www.ora.com/catalog/javadata/

[25] Net Communications Ltd., *"JDBC Explained"*
http://pw2.netcom.com/ wbillow/java/jdbc.tml

[26] Symantec Corp., *"Symantec's new dbAnywhere"*
http://cafe.symantec.com/dba/index.html

[27] Imaginary Corp., *"Imaginary JDBC Solutions"*
http://www.imaginary.com/Java/

[28] DataRamp Corp., *"DataRamp JDBC Products"*
http://www.dataramp.com/

[29] Nial McKay, *"RMI's Future cloudy as Sun Tries to Please CORBA Partners"*
Java World, IDG's Magazine for Java Community, June 1997.
http://www.javaworld.com/javaworld/jw-06-1997/jw-06-idgns.rmi.html

[30] Silicon Valley Corp., *"Java SIG JDBC Talk Page"*
http://www.sug.com/java-sig.html

[31] Caribou Lake Software, *"RMI versus CORBA"*
http://www.cariboulake.com/techinfo/rmi_corba.html

[32] JavaSoft Inc., and Intersolv Corp., *"JDBC Test Tool"*
ftp://splash.javasoft.com/pub/JDBCTest1_0*3.zip*

[33] Simba Technologies , *"Sneak Preview of JDBC Connectivity"*
http://www.simbatech.com/products/jdbc.html

[34] Jerrid R. Hamann, Maters Thesis, *"Analysis of JDBC & its applications in a multi-platform, multi-DBMS environment"*
Texas A&M University, College Station, Texas.

[35] Visigenic Corp., *"Developing Database Independent Applications with ODBC"*
http://www.visigenic.com/info/odbc.html, May 1996.

[36] *"RMI versus CORBA"*
http://www.fivepoints.com/ajug/info/tech/rmi/

[37] Nial McKay, *"JavaSoft to solve RMI's interoperability problem – Company stands behind RMI, integrates technology with CORBA*
Java World, IDG's Magazine for Java Community, July 1997.
http://www.javaworld.com/javaworld/jw-07-1997/jw-07-idgns.rmi.html

[38] MicroSoft Corp., *"Microsoft Visual J++: The Fast Track to Power Java Programming"*
http://www.microsoft.com/visualj/, July 1996.

[39] MicroSoft Corp., *ODBC - Open Database Connectivity Overview"*
http://198.105.232.6/KB/dskapps/word/Q110093.htm, Feb 1996.

[40] George Reese, *"George Reese's Java Pages"*
http://u r t . : r e g i b r a r y.com/borg/Java/

[41] NCST, *"Databases and Information Systems"*
http://www.ncst.ernet.in/dbois/jdbc/index.html

[42] Intersolv Corp., May 1996, *"Intersolv's Whitepaper on JDBC-ODBC"*
http://www.intersolv.com/programs/sweet$_w$*hite.htm*